Designing software interfaces

Andy Buckley
University of Glasgow

MCnet Computing School, Mariaspring, 18 May 2016

Intro

- Design is the best bit! Chance to pour all your creativity and experience into making the interface you want
- It's satisfying: craftsmanship, a bit of artistry, applied psychology, …
- *But it's hard:* need to think 3 steps ahead, and keep many things in mind
- And leave yourself room to maneuvre: you *will* get some things wrong, so how to avoid getting locked into bad ideas?
- Interfaces come in many forms: CLI, GUI, API, data formats
- Many idioms, mottos, proverbs, parables…
- Don't fall for "not invented here" disease

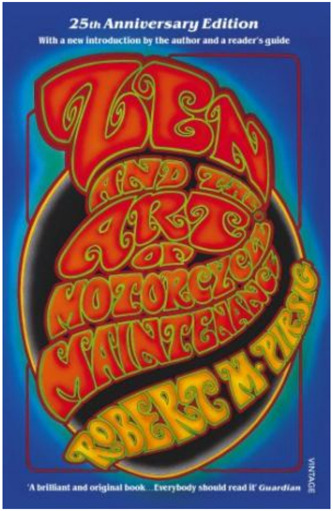
Think of the user first

- The designer mindset: thinking as a user
  - You will need to agonise, sweat, and write clever (or repetitive) code to make it *seem* simple to your users
  - cf. Rivet's projections, cuts & analysis loader systems
  - Ideally they will have no idea how much is going on below the surface
- "Make simple things simple; and difficult things possible"
- **Metaphors are key:** a good metaphor means that an interface "flows"…and less docs needed!
- **Consistency** ⇒ "Principle of least surprise"

Quality

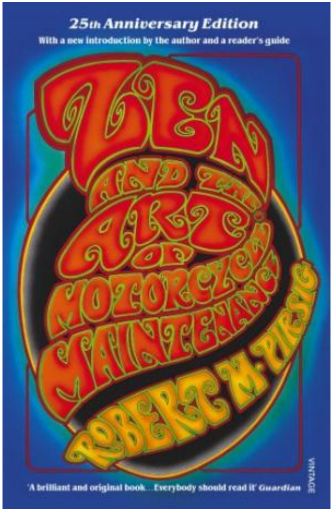
Emphasise *quality*:
take pride in getting it right

- Implies iteration: "Build one to throw away – you will anyway"
- Plan for wrong turns ⇒ pro-active versioning and deprecation strategies
- Defensive design ⇒ caution, think ahead, and be prepared for extra up-front work


25th Anniversary Edition
With a new introduction by the author and a reader's guide
Zen and the Art of Motorcycle Maintenance
Robert M. Pirsig
'A brilliant and original book... Everybody should read it' Guardian
VINTAGE

Quality

- $\Rightarrow$ Avoid short-cuts…
  except when you don't!
  - Don't forget to be proportionate: if you're really hacking a one-off, do what gets a good enough result quickly
  - No point in heavily engineering a throw-away
  - But as you get better, robust designs will be familiar and not take significantly longer


25th Anniversary Edition
With a new introduction by the author and a reader's guide
Zen and the Art of Motorcycle Maintenance
Robert M. Pirsig
'A brilliant and original book…Everybody should read it' Guardian
VINTAGE

Caring

What qualities do really bad programmers share? (Quora)

They don't care.

- They don't care about the reason why they're coding – the thing they're supposed to make work
- They don't care about how they go about coding – best practices, methodology, modularization, architecture...
- They don't care about understanding what needs to be done before they start coding – they jump right in
- They don't care about the language they're going to use – anything goes, the looser the better
- They don't care about putting in the work that's needed to build things the right way – they'll take shortcuts just because
- They don't care about tracking their progress and making sure others can understand what they're doing – after all, bug trackers are only needed when your code has bugs, right?
- They don't care about writing tests and documentation – if you want to understand what the code does, why don't you just look at it?
- They don't care about your advice nor your opinion, because, quite frankly, they already know better

“Methodologies”

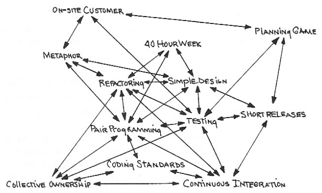
First, collect requirements. Think before you act & have a plan. Then…

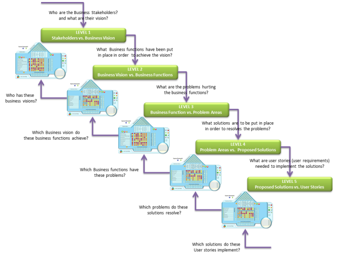
- **Lone genius:** not entirely flippant. Single-minded focus can be key to design coherence
  - Especially in HEP, everyone has met some system “designed by committee”
  - But you *will* need to work with others
- **Discussion / prototyping:**
  - Get feedback early
  - Linus’ Laws: “Release early, release often”; “With many eyeballs, all bugs are shallow”
  - Card-passing exercises for OO design

"Methodologies"

First, collect requirements. Think before you act & have a plan. Then…

- **"Agile" / "Extreme" / "SCRUM"**
  - Lightweight, minimal spec'ing, pair programming & code review
  - YAGNI – You ain't gonna need it is a good principle
  - As with anything, can be taken too literally: don't skimp on quality and metaphor soundness
  - But a good motto when deciding whether to add obscure features "for completeness / for symmetry / because I know how" (a very physicist trait)
- **"Patterns"** – urgh. Better to *understand*, not apply one-size-fits-all





Who are the Business Stakeholders? and what are their vision?
LEVEL 1 Stakeholders vs. Business Vision
What Business functions have been put in place in order to achieve the vision?
LEVEL 2 Business Vision vs. Business Functions
What are the problems hurting the business functions?
LEVEL 3 Business Function vs. Problem Areas
What solutions are to be put in place in order to resolves the problems?
LEVEL 4 Problem Areas vs. Proposed Solutions
What are user stories (user requirements) needed to implement the solutions?
LEVEL 5 Proposed Solutions vs. User Stories
Who has these business visions?
Which Business vision do these business functions achieve?
Which Business functions have these problems?
Which problems do these solutions resolve?
Which solutions do these User stories implement?

APIs

Application programming interfaces – i.e. code interfaces

- Are you writing a library or an executable? *Why not both?*
- Many complex problems are made worse by misalignment of problem and code design
  - Think, think, think
  - Prototype (maybe in Python, even for C++ projects), and keep going until you feel it click
  - Try writing your fantasy ideal user script, then make the library that enables it to work!
- **Identify what is the "atomic"** ***key object*** **in the problem**
  - This may be a single thing or a collection of them, it depends on the main use-case
  - But single objects make your code more flexible, repurposeable
  - Cf. LHAPDF 5 $\rightarrow$ 6 key object from PDF set $\rightarrow$ PDF member. Or Professor 1 $\rightarrow$ 2: key object from histogram $\rightarrow$ single interpolated value
  - **Divide and conquer** along the lines of natural object relationships
  - "Keep an open mind but not so open that your brain falls

APIs (2)

Application programming interfaces – i.e. code interfaces

- Be careful: *it's far easier to add than to take away*
  ⇒ YAGNI and deprecation strategies
  - Plan for change, and be prepared for the pain of removing bad ideas rather than freezing them in forever
  - But minimise them: discuss before release, use betas
- Use OO well: it can help a lot…
  but not everything should be (in) a class!!
  - In parallel code, objects and methods can be a nightmare :-(
    That's a design requirement, too
  - Hybrid designs, e.g. "object views" on to ensemble arrays can work, but *more moving parts* = *more failure modes*
  - KISS – "Keep it simple, stupid"
- Do you get the feeling that the tail is wagging the dog? Maybe the type system is forcing you to create extra layers of hierarchy? Can you be smart and beat it? Use that challenge to inspire

Coding convention

- Choosing good names is *really* important. Way more than you think
- Use the style convention to minimise user/maintainer need to chase down definitions
- Micro-consistency is worthwhile: e.g.
  - "**`mk`**" or "**`make`**"?
  - **`camelCase`** or **`under_scores`**?
  - **`CONSTANTS`**, **`Classes`**, **`functions`**. (+ **`variables`**, internally)
  - Clearly mark private class members (I use Python leading underscore everywhere; YMMV)
  - Pass common arguments in the "same order" everywhere (or $\Rightarrow$ single object)
  - Attention to detail pays off in *lack of surprise*

Coding convention

- Distinction between this and *style*
  - Whitespace is not an API thing – but choose a decent, widespread convention and *be consistent*
  - And respect other people's conventions when working on their code
  - Great opportunities for bikeshedding – **STOP IT!**

C(++) specifics

- **C++ was not designed to minimise coupling, provide clean interfaces, etc.**
- What to expose? Not everything should have a public header
- *"Does this code unit need a header at all?"*
- Use namespaces, forward declarations, and avoid `using` in headers – unless namespaced
- C++ include system is broken: exposes implementation details via memory layout of internal data, forces wide rebuilds
    - Strategies, e.g. PIMPL method to decouple dependencies
- Pointers as a hint of memory management; references otherwise.
    - I like to use "mk" names when memory is allocated, as a signal for the user to watch out

Object orientation

- Object orientation is about polymorphism and encapsulation first: "need to know"
- In many situations it is a perfect match to the problem: interacting self-contained objects are the natural picture
- Avoids the need for developer omniscience ⇒ maximum-entropy code problem
- But **not *everything* needs to be an object!**
  - This was the error of Java, "the OO language"
  - Parables: object-oriented toaster (not every logical base class is necessary), kingdom of nouns (not everything is an object)
  - Keep it flexible; mix and match styles; Zen/kung-fu pastiche x yet y

Sooooo bad…

Object orientation

- You do not need to use all the language features! Think of the user…
- Are objects compatible with scalability, and parallel/vectorised performance requirements?
    - Encapsulation is conceptually great for the programmer, potentially fatal for CPU caches
    - "Views" on to contiguous data arrays may work (are there OO langs that help?!)
    - **It depends** on how your users will use your tools

Sooooo bad…

Documentation

- Everyone hates writing documentation (and tests) – but if your interface makes intuitive sense, you need less of it
- Document as you go along, make it a reflex action so you can't *not* document! And apply quality standards to that, too
- "Out-of-date documentation is worse than no documentation"
- Reduce barriers ⇒ Doxygen and Sphinx, doctest
- Can integrate "traditional documentation" with building the project website in these tools
- Self-documenting code structure, cf. queriable Rivet analysis metadata:
  - Build documentation *using* the API in addition to "code scrapers"

Command-line interfaces

- Unix philosophy of "small tools, each doing one job well" is excellent advice – also in APIs
- Split applications into separated components. Communication by pipes, sockets, etc. – use the road-tested OS features
- Interface consistency again!
- Names: avoid underscores, avoid file extensions
- Start commands with a common prefix to allow "tab searching"
- Order script names, data files, etc. for convenient alphanumeric grouping when listed
- Build in looping potential via "last-arg = open-ended list" $\Rightarrow$ saves users need to write shell loops
- Use standard option parsing libraries e.g. Python ~~optparse~~ argparse
  - Note: not everything needs a short **-x** option. Don't waste them!

Data format design

- I'm a big fan of plain text for non-huge data
  - Robustness, human readability/debuggability
  - Space efficiency can be traded
  - Parsing efficiency can be improved with some tricks
- Base on standard meta-formats as much as posssible: YAML, JSON, (XML), HDF5, (ROOT), ...
- Think again about how you'll handle changing requirements: include versioning from the beginning, to handle "schema evolution"
- Steering/config file formats and user interfaces suit restricted subsets of usage possibility: don't be tempted to make a Turing-complete CLI or config syntax. For "non-standard" uses, tell your users to code exactly what they want via your friendly, flexible, and powerful API!